DESENVOLVIMENTO DE UM JOGO 2D COM O FRAMEWORK PHASER

mpsxdeveloper
https://github.com/mpsxdeveloper

Resumo

Esse artigo demonstra a construção de um simples jogo 2D com o Phaser, explicando como será a lógica do jogo e analisando algumas das principais funções embutidas nesse framework e os códigos mais importantes que foram utilizados para a conclusão desse projeto.
Palavras-chave: Jogo. Phaser. Framework.

Abstract

This article demonstrates building a simple 2D game with Phaser, explaining how the game logic will be and analyzing some of the main functions built into this framework and most important codes that were used to complete this project.
Keywords: Game. Phaser. Framework.

Introdução

Phaser é um framework de código aberto que serve para o desenvolvimento de jogos para navegadores web móveis ou desktop. É rápido, leve, fácil de utilizar, extensível com o uso de plugins e possui muitas funções para criar diversos recursos nos jogos de forma simples, tais como elementos de física, animações, partículas, colisões, câmera, sons, mapas, etc. É uma das principais ferramentas utilizadas para o desenvolvimento de jogos atualmente.

As mecânicas e estrutura do jogo desenvolvido

O jogo consiste em um círculo controlado pelo jogador através do teclado (teclas direita e esquerda). Inicialmente o círculo é posicionado na parte superior da tela, e quando o jogo começa, a gravidade do jogo puxa o círculo para baixo. No cenário vão surgindo dois tipos de figuras de forma quadrada, nas quais o círculo pode colidir e gerar um enfeito de rebatida, fazendo com que o círculo seja jogado para cima novamente. O jogo encerra quando o círculo atinge o limite inferior do cenário, de modo que quanto mais o jogador manter o círculo flutuando, maior será sua pontuação, que é incrementada através de um contador fixo.

Configuração do ambiente de desenvolvimento

Foi utilizado o editor de códigos Visual Studio Code, porém outros editores de código podem ser utilizados sem problema, pois se trata de uma estrutura de projeto simples. A estrutura de pastas do projeto consistem na pasta raiz *phaser-2D-game*, uma pasta com o nome *assets*, onde ficam as imagens utilizadas no jogo, e na raiz do projeto ficam os arquivos *game.js* (responsável pela lógica

do jogo) e *index.html* (a página que exibe o jogo). A figura 1 a seguir demonstra como está montada a estrutura final dos arquivos do projeto:

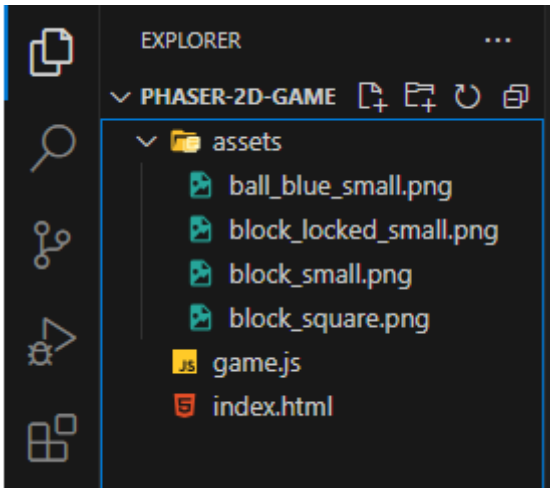

Figura 1: Estrutura do projeto

Os assets dos jogos (as imagens utilizadas) são do site Kenney, um site com assets gratuitos para jogos, acessado no endereço <https://www.kenney.nl/>. Foi utilizado o conjunto Rolling Ball Assets, acessado da página <https://www.kenney.nl/assets/rolling-ball-assets>.
A página *index.html* possui a seguinte codificação:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="//cdn.jsdelivr.net/npm/phaser@3.70.0/dist/phaser.min.js"></script>
    <title>Phaser 2D Game</title>
</head>
<body>
    <div id="game" style="margin: 0 auto; width: 600px;"></div>
    <script src="game.js"></script>
</body>
</html>
```

A página *index.html* que exibe o jogo é bem minimalista, onde no cabeçalho a biblioteca Phaser é chamada através de uma requisição para uma CDN. No corpo da página temos apenas uma DIV centralizada, que é o lugar onde o Phaser vai carregar e executar o jogo, e logo em seguida temos a chamada para o script *game.js*, que é a lógica e todo código do jogo feito com Phaser.

Explicando o código do jogo

O código do script game.js inicia com a seguinte configuração:

```
const config = {
    type: Phaser.AUTO,
    width: 600,
    height: 500,
    parent: 'game',
    backgroundColor: '#CCCCFF',
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },
            debug: false
        }
    },
    scene: {
        preload: preload,
        create: create,
        update: update
    }
}

const MAX_RIGHT = 500
const MIN_LEFT = 100
const VELOCITY = 15
let increase = 0

const game = new Phaser.Game(config)
```

Cria-se uma variável *config* com diversas configurações que deverá ter o jogo. Em Phaser.AUTO, deixamos a cargo do Phaser decidir qual é a forma de renderização do jogo, que pode ser Canvas ou WebGL. Os atributos *width: 600* e *height: 500* são a largura e altura do canvas utilizado para desenhar o jogo (valores que combinam exatamente com o tamanho do DIV de *index.html*). Em *physics* determinamos qual será o tipo de física aplicada aos objetos do jogo, assim como determinamos o valor de 300 para a gravidade em *y*. Ainda em *config*, no objeto *scene*, informamos ao Phaser quais serão os métodos para as funções principais de *preload*, *create* e *upgrade*.
As variáveis constantes MAX_RIGHT e MIN_LEFT são os valores limites da posição x dos quadrados que surgem na tela. VELOCITY é o valor inicial fixo da velocidade de movimentação dos quadrados na tela, que somados com a variável *increase* vão alterando a velocidade final do movimento. Com tudo isso, podemos iniciar de fato uma instância do Phaser passando a variável *config* como parâmetro. Em seguida temos a função *preload*:

```
function preload() {
    this.load.image('player', 'assets/ball_blue_small.png')
    this.load.image('yellow_block', 'assets/block_locked_small.png')
    this.load.image('green_block', 'assets/block_small.png')
}
```

A função preload é responsável por carregar todos os assets que serão utilizados pelo jogo, que nesse exemplo são apenas três imagens, mas poderiam ser dezenas de imagens, sons, mapas, coleções de dados em formato JSON, dentre outros tipos de arquivos. Cada arquivo carregado tem um nome atribuído a ele que deve ser único, pois esse arquivo deverá ser utilizado em outros trechos de código do jogo.
A função *create* vem a seguir e possui o código abaixo:

```
function create() {
    score = 0
    running = false
    platforms = this.physics.add.group({
        immovable: true,
        allowGravity: false
    })
    block_small = this.add.sprite(300, 32, 'green_block').setOrigin(0, 0)
    platform_green1 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 100, 'green_block').setOrigin(0, 0)
    platform_yellow1 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 175, 'yellow_block').setOrigin(0, 0)
    platform_green2 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 250, 'green_block').setOrigin(0, 0)
    platform_yellow2 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 325, 'yellow_block').setOrigin(0, 0)
    platform_green3 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 400, 'green_block').setOrigin(0, 0)
    platform_yellow3 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 475, 'yellow_block').setOrigin(0, 0)
    platform_green4 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 550, 'green_block').setOrigin(0, 0)
    platform_yellow4 = this.add.sprite(Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT), 625, 'yellow_block').setOrigin(0, 0)
    platforms.add(platform_green1)
    platforms.add(platform_yellow1)
    platforms.add(platform_green2)
    platforms.add(platform_yellow2)
    platforms.add(platform_green3)
    platforms.add(platform_yellow3)
    platforms.add(platform_green4)
    platforms.add(platform_yellow4)
    score_text = this.add.text(5, 5, "Score: " + score, {fontSize: 38, color: "#000000", fontStyle: "bold"})
    information_text = this.add.text(200, 210, "READY?", {fontSize: 58, color: "#000000", fontStyle: "bold italic"})
    player = this.physics.add.sprite(300, 1, 'player').setOrigin(0, 0)
    player.body.setAllowGravity(false)
```

```
    player.setBounce(1)
    player.setCollideWorldBounds(true)
    this.physics.add.collider(player, platforms)
    cursors = this.input.keyboard.createCursorKeys()
    this.loopEvent = this.time.addEvent({ delay: 500, callback: onLoopEvent, callbackScope: this,
loop: true })
    this.startEvent = this.time.addEvent({ delay: 3000, callback: onStartEvent, callbackScope: this,
loop: false })
}
```

A variável *score* salva a pontuação do jogador e a variável *running* determina se o jogo está rodando ou não, iniciando como falso. Em seguida é criado um grupo de objetos chamado *platforms*, onde guardará todos os sprites dos quadrados do jogo (que são as plataformas). Foi setado *immovable: true* porque os sprites desse grupo não precisam reagir ao impacto da colisão com o círculo, apenas devem continuar subindo na tela e *allowGravity: false* porque sem isso as plataformas iam cair da mesma forma que o círculo.
Em seguida temos *block_small = this.add.sprite(300, 32, 'green_block').setOrigin(0, 0)*. Esse bloco serve somente para dar a impressão que está impedindo o círculo de cair enquanto não começa o jogo, porém esse sprite foi criado sem física, ou seja, é apenas uma imagem na tela sem interação, fora isso ele segue o padrão de criação dos outros sprites com física, onde é passado as coordenadas iniciais x e y na tela, qual a imagem associada a esse sprite (*'green_block'* foi criado na função preload) e *setOrigin(0, 0)*. Por padrão as imagens ou sprites são criados com referência de posição no centro da imagem, mas podemos mudar isso com *setOrigin*, informando os valores que queremos, geralmente as coordenadas x e y (0, 0) (canto superior esquerdo) são valores de um padrão que estamos acostumados a trabalhar como referência em jogos 2D.
Seguindo esse raciocínio, foram criados mais 8 blocos utilizando os sprites *'green_block' e 'yellow_block'*, com posições y fixas, de modo que houvesse espaço entre cada bloco para que o círculo pudesse se movimentar (lembre que o círculo terá o efeito rebatida ao colidir com as plataformas), porém o valor da posição x de cada bloco é gerada aleatoriamente usando a função interna *Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT)*. Os valores x mínimo e máximo devem pertencer a uma área razoável para que não fique impossível de jogar ou muito difícil logo de começo. Esses valores foram estipulados por constantes criadas anteriormente, pois a vantagem do uso de constantes é poder modificar os valores mínimo e máximo de x para os blocos em apenas um lugar, muito útil para testes e manutenção do jogo.
O próximo passo foi adicionar os 8 blocos na variável de grupo *platforms* com o *platforms.add*. Ressalta-se que embora os sprites dos blocos tenham sido criados sem física embutida, o grupo *platforms* foi criado com física, portando seus elementos terão uma física padrão. A vantagem de usar grupos para gerenciar sprites é que é possível alterar o comportamento de cada sprite através de loops na variável de grupo em vez de fazer isso em cada sprite.
Em seguida adicionamos duas variáveis de texto, que são *score_text* e *information_text*, para imprimir na tela a pontuação do jogador e mensagens de início e fim de jogo. Possuem argumentos de posição x e y iniciais, tais como os sprites, o texto a ser exibido, e um objeto opcional contendo configurações de fonte e estilo do texto. Todos esses atributos podem ser alterados posteriormente.
O último sprite a ser criado é o sprite do círculo que o jogador controla. Sendo criado por último, ele ficará na frente dos outros sprites caso se movimente através deles. Note que ao criarmos o sprite com a linha *player = this.physics.add.sprite(300, 1, 'player').setOrigin(0, 0)* estamos

atribuindo física ao sprite, e como o jogo foi configurado para ter gravidade, assim que o sprite é criado ele será puxado para baixo, porém, logo abaixo temos a instrução *player.body.setAllowGravity(false)* o que desabilita a força de gravidade no sprite, deixando ele estático.
Em *player.setBounce(1)* configuramos a força com que o sprite vai ter no efeito rebatida, o valor 1 é o valor máximo (faça testes de valores decimais entre 0 e 1 para ver como se comporta o efeito rebatida no círculo) onde 0 é sem efeito nenhum. No trecho *player.setCollideWorldBounds(true)* definimos que o sprite *player* (círculo) não deve ultrapassar os limites de largura e altura estabelecidos em *width* e *height* no objeto *config* do início do código. Isso nos permite movimentar o sprite sem nos preocupar com testes e ajustes das suas posições x e y. Seguindo adiante, temos o código *this.physics.add.collider(player, platforms)* onde informamos que o círculo deve colidir com os blocos contidos no grupo platforms. Sem isso, as colisões seriam ignoradas.
Na linha *cursors = this.input.keyboard.createCursorKeys()* configuramos um acesso rápido para a leitura de algumas teclas comuns usadas como input em jogos utilizando o teclado, que são as teclas direita, esquerda, cima, baixo, shift e barra de espaços.
Concluindo a função *create*, temos *this.loopEvent = this.time.addEvent({ delay: 500, callback: onLoopEvent, callbackScope: this, loop: true })* que é a criação de um evento que será rodado em loop, com uma frequência de 500 milissegundos, que invoca a função *onLoopEvent* a cada loop. Em seguida, temos *this.startEvent = this.time.addEvent({ delay: 3000, callback: onStartEvent, callbackScope: this, loop: false })* que invoca a função *onStartEvent* após 3 segundos, porém essa chamada foi configurada como *loop: false*, portanto só acontecerá apenas uma vez.
Observa-se agora a função *update*, que possui a seguinte lógica:

```
function update() {
    if(running) {
        if(cursors.left.isDown) {
            player.setVelocityX(-70)
        }
        else if(cursors.right.isDown) {
            player.setVelocityX(70)
        }
        if(player.y > 500 - (player.height + 1)) {
            player.body.stop()
            running = false
            player.setBounce(0)
            player.tint = 0xff0000
            information_text.setText("GAME OVER").setFontSize(58).setFontStyle("bold
italic").setColor("#ff0000").setX(150).setTint(0xff00ff, 0xffff00, 0x00ff00, 0xff0000)
        }
    }
}
```

A função update fica rodando em loop, e é gerenciada pelo Phaser, nesse método deve-se inserir a lógica principal do jogo, a leitura das teclas pressionadas e gerenciar os aspectos principais da gameplay. Note que todo o código de update só é executado se *running* for igual a true, pois só faz sentido esses testes e mudanças do jogo serem verificados enquanto o jogo não estiver encerrado ou

pausado. Sendo assim, é verificado quando o jogador pressiona as teclas direita e esquerda, movimentando o círculo com a função *player.setVelocityX*, passando valor positivo para movimentar para a direita, e valor negativo para movimentar para a esquerda. Como foi configurado para que o sprite do círculo não ultrapasse os limites da tela do jogo, não se faz necessário aqui checagem extra da posição atual do círculo.
Aqui também é checado se o círculo atinge a parte inferior do limite da tela, nesse caso, é considerado jogo terminado, então imediatamente com o código *player.body.stop()* zeramos a aceleração e velocidade do sprite. Também é setado *running = false*, para que os blocos parem de se movimentar e não seja mais feita leitura das teclas. Também configura-se *player.setBounce(0)* para que cesse o efeito de rebatida do sprite, e pinta-se o sprite com uma cor vermelha como feedback. E por último atualizamos a variável *information_text* para imprimir aproximadamente no centro da tela a mensagem de jogo terminado.
Prosseguindo, a função *onLoopEvent* possui o código demonstrado a seguir, lembrando que essa função é chamada a cada 500 milissegundos:

```
function onLoopEvent() {
   if(running) {
      score += 1
      score_text.text = "Score: " + score
      Phaser.Actions.Call(platforms.getChildren(), function(platform) {
         platform.y = platform.y - (VELOCITY + increase)
         if(platform.y < -platform.height) {
            platform.x = Phaser.Math.RND.between(MIN_LEFT, MAX_RIGHT)
            platform.y = 500 + platform.height
         }
      }, this)
      if(score % 50 === 0) {
         increase += 1
      }
   }
}
```

O código verifica se *running* é igual a true, sendo assim, acrescenta 1 unidade ao score do jogador e atualiza essa informação na tela, com a variável *score_text*. Em seguida, é feito um loop em todas os blocos da variável *platforms*, com a chamada *Phaser.Actions.Call(platforms.getChildren(), function(platform)*, e dentro do loop, atualizamos a posição y de cada bloco com somatório de (VELOCITY + increase). Esse valor vira negativo de modo que diminuímos o valor de y do bloco em relação à sua posição atual, fazendo com que o bloco suba na tela. Se o objetivo fosse que o bloco descesse, esse valor teria que ser positivo. Porém, o bloco não deve ficar subindo infinitamente, portanto faz-se um teste para saber se ultrapassou o limite superior da tela, caso isso aconteça, é criada uma nova posição aleatória na coordenada x e uma posição fixa em y (um pouco abaixo do limite inferior da tela), e o bloco é reposicionado, subindo novamente na tela em loop. E para concluir essa função, a cada 50 pontos aumenta-se a variável increase em 1 unidade, isso fará com que a cada 50 pontos, os blocos subam um pouco mais rápido, criando um fator de dificuldade para o jogo.
Para a última função, *onStartEvent*, tem-se o código abaixo:

```
function onStartEvent() {
    information_text.text = ""
    block_small.destroy()
    running = true
    player.body.setAllowGravity(true)
}
```

Essa função é chamada apenas uma vez no início do jogo. Ela apaga a mensagem “Ready?” da tela, destrói o bloco que “segura” o sprite do círculo, seta a variável *running* para *true*, o que dá início ao jogo de fato e por fim, com *player.body.setAllowGravity(true)* a gravidade é liberada para o círculo, fazendo com que ele caia imediatamente, em condições de ser movimentando pelo jogador. A figura 2 demonstra como ficou a tela de jogo desse projeto:

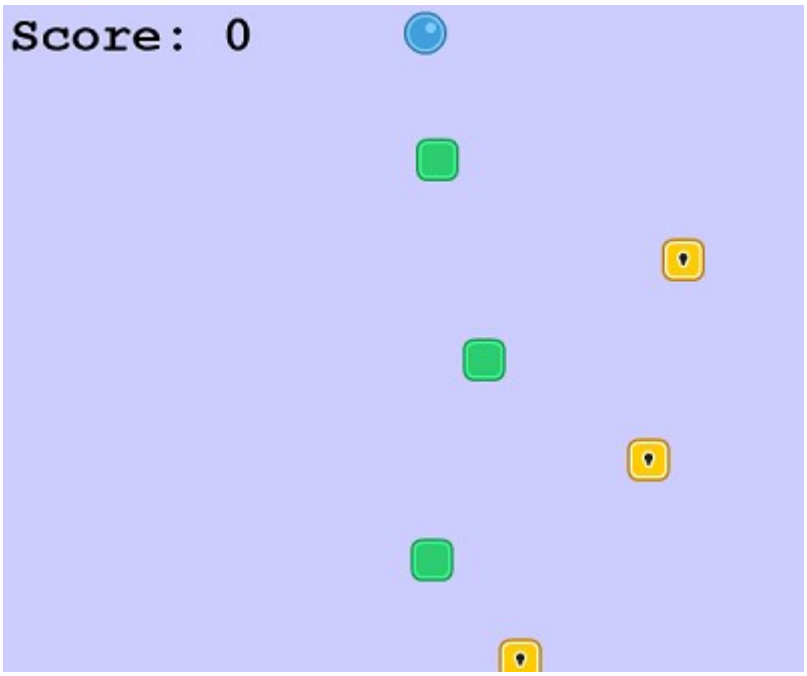

Figura 2: Tela do jogo

Considerações finais

Phaser tem se consolidado durante os últimos anos como uma das melhores opções para desenvolvimento de jogos para navegadores desktop ou mobile. O jogo desenvolvido para esse artigo utilizou recursos práticos desse framework com funções prontas para diversas situações de gameplay, assim como a reconfiguração de elementos do jogo de forma facilitada pela ferramenta. Conforme o aprendizado do framework for se expandindo, pode-se aprimorar ainda mais o jogo com mais recursos e elementos, além de criar outros tipos de jogos com mais complexidade.

Bibliografia

Phaser 3 Examples. Disponível em: <https://phaser.io/examples/v3>. Acesso em: 22/01/2024.